Vivekanand Education Society's
Institute of Technology
Approved by AICTE & Affiliated to University of Mumbai

Artificial Intelligence and Data Science Department

Deep Learning/Odd Sem 2023-23/Experiment 2a

Name : Rupesh Dhirwani	**Class/Roll No. :** D16AD - 10	**Grade :**

Title of Experiment : To apply appropriate learning algorithms to learn the parameters of the supervised single layer feed forward neural network.

Objective of Experiment : The objective of this study is to select and implement an appropriate learning algorithm for effectively training the parameters of a supervised single-layer feedforward neural network.The aim is to optimize the network's ability to capture and learn patterns within the provided training data. By systematically evaluating and comparing various learning algorithms, we seek to identify the method that best facilitates parameter convergence, enhances predictive accuracy, and accelerates learning efficiency within the context of a single-layer feedforward neural network.

Outcome of Experiment : Upon successfully achieving the stated objective, we anticipate identifying a learning algorithm that demonstrates superior performance in training the parameters of the supervised single-layer feedforward neural network. This algorithm is expected to showcase rapid convergence, enabling the network to efficiently learn from the training data and adapt its parameters to capture underlying patterns. Consequently, the neural network's predictive accuracy is likely to be significantly improved, resulting in more accurate and reliable predictions on unseen data.

Problem Statement : This problem demands a comprehensive evaluation of various learning algorithms, considering factors such as convergence behavior, computational efficiency, adaptability to different data types, and generalization to new data points. Addressing this problem is essential not only for improving the effectiveness of single-layer feedforward neural networks but also for advancing our understanding of machine learning techniques across diverse applications.

Deep Learning/Odd Sem 2023-23/Experiment 2a

Description / Theory :

STOCHASTIC GRADIENT DESCENT (SGD) : -

In the intricate realm of deep learning, Stochastic Gradient Descent (SGD) emerges as a cornerstone optimization algorithm, fine-tuned to navigate the complexities of training neural networks. Beyond its efficiency, SGD introduces a controlled randomness that shapes the learning dynamics of the optimization process.

At its core, SGD diverges from conventional Gradient Descent by leveraging mini-batch processing. Instead of computing gradients for the entire dataset, SGD samples subsets, or mini-batches, thereby drastically reducing the computational burden per iteration. This nimble approach facilitates frequent parameter updates, accelerating convergence and enhancing the optimization process.

The introduction of randomness in SGD's mini-batch processing, while boosting efficiency, introduces both challenges and opportunities. On one hand, this controlled randomness could lead to a noisy gradient estimate, causing oscillations in the optimization path.

On the other hand, this very noise helps the algorithm escape local minima and saddle points, enhancing its ability to discover more global minima. Balancing these dynamics necessitates thoughtful parameter tuning. Learning rate scheduling and regularization techniques play crucial roles in ensuring SGD's convergence and preventing divergence. Furthermore, adaptive learning rate methods like Adagrad, RMSProp, and Adam have evolved from SGD, addressing its limitations and providing mechanisms to dynamically adjust learning rates based on historical gradient information.

MINI-BATCH GRADIENT DESCENT:-

Mini-Batch Gradient Descent is a vital optimization technique that combines the advantages of both Stochastic Gradient Descent (SGD) and Batch Gradient Descent. In the world of deep learning, where training data can be massive, this method strikes a balance between computational efficiency and convergence speed.

Unlike traditional Batch Gradient Descent, which processes the entire dataset in each iteration, Mini-Batch Gradient Descent works with smaller subsets, or mini-batches. This approach reaps the benefits of both worlds: it leverages randomness from SGD and minimizes oscillations from Batch Gradient Descent.

By choosing an appropriate mini-batch size, the algorithm optimizes the trade-off between computational efficiency and convergence stability. Smaller mini-batches lead to faster updates, as each iteration requires less computation. Mini-Batch Gradient Descent offers several advantages. It effectively harnesses parallelism, allowing the exploitation of modern hardware architectures.

This accelerates training by performing computations simultaneously on multiple samples. Additionally, Mini-Batch Gradient Descent provides smoother convergence compared to SGD, owing to the larger sample size that reduces noisy gradient estimates.

In practice, mini-batch size selection depends on factors such as dataset size, available resources, and desired convergence behavior. In conclusion, Mini-Batch Gradient Descent stands as a versatile optimization approach, offering a middle ground between stochastic and batch processing.

Its adaptability to different problem scenarios and computational capabilities makes it an essential tool for efficiently training complex neural networks and driving advancements in the field of deep learning.

ADAGRAD:

In the landscape of optimization algorithms for deep learning, Adagrad, short for Adaptive Gradient Descent, is a trailblazing technique that introduces adaptability to the learning rate.

This innovation addresses the challenge of learning rates, offering a dynamic approach that tailors the step size for each parameter during training. Traditional gradient descent algorithms employ a single learning rate for all parameters, which can lead to suboptimal convergence in the presence of varying gradients.

Adagrad counters this by adjusting the learning rate for each parameter individually based on the historical information of the gradient magnitudes. The magic of Adagrad lies in its personalized approach to learning rates. Parameters that have larger gradient magnitudes receive a smaller learning rate, allowing them to take smaller steps and avoid overshooting.

Conversely, parameters with smaller gradients receive larger learning rates, enabling faster convergence along flatter dimensions of the optimization landscape. The adaptability of Adagrad makes it particularly well-suited for non-convex optimization problems commonly encountered in deep learning. It shines in scenarios where gradients vary significantly across different parameters, helping the optimization process converge more efficiently.

However, Adagrad's effectiveness isn't universal. Over time, the accumulation of squared gradients in the denominator of the learning rate formula can lead to diminishing updates, causing the learning rate to become excessively small. This can hinder further convergence, making Adagrad less suited for long training sessions or situations where the optimization landscape changes dynamically.

To mitigate this limitation, variations of Adagrad have been developed. Algorithms like RMSProp and Adam (Adaptive Moment Estimation) address the diminishing learning rate issue by introducing mechanisms to control the historical accumulation of squared gradients. In essence,

Deep Learning/Odd Sem 2023-23/Experiment 2a

Adagrad emerges as an innovation that injects adaptability into gradient descent optimization. By dynamically tuning learning rates for each parameter, it navigates the complexities of the optimization landscape in deep learning, optimizing convergence in scenarios where standard gradient descent algorithms falter.

Algorithm / Pseudo Code:

Stochastic Gradient Descent (SGD):-

- Define a method named `gradient_descent` inside the `LogisticRegressionSGD` class.
- Accept input data `X`, target labels `y`, learning rate `learning_rate`, and the number of epochs `epochs` as arguments.
- For each epoch in the range of `epochs`:
- Iterate over each data point in the training dataset (indexed by `i`).
- Predict the output using the `predict` method of the logistic regression model.
- Calculate the error by subtracting the predicted value from the true label (`y[i] - prediction`).
- Update the model's weights using the gradient descent formula: `self.weights += learning_rate * error * X[i]`.
- Update the model's bias using: `self.bias += learning_rate * error`.

Setting SGD Hyperparameters:-

- Define the hyperparameters `learning_rate_sgd` and `epochs_sgd`.

2. Mini-Batch Gradient Descent:-

- Define a method named `gradient_descent` inside the `LogisticRegressionMiniBatch` class.
- Accept input data `X`, target labels `y`, learning rate `learning_rate`, number of epochs `epochs`, and batch size `batch_size` as arguments.
- For each epoch in the range of `epochs`: - Iterate over data points in mini-batches of size `batch_size`.
- Extract a batch of input features `batch_X` and corresponding target labels `batch_y`.
- Calculate predictions using the logistic function, rounded to binary values.
- Calculate error by subtracting predicted values from true labels (`batch_y - prediction`).
- Update model weights using the gradient descent formula: `self.weights += learning_rate * np.dot(batch_X.T, error) / batch_size`.
- Update model bias using: `self.bias += learning_rate * np.sum(error) / batch_size`.

Setting Mini-Batch Hyperparameters:-

- Define the hyperparameters `learning_rate_mini_batch`, `epochs_mini_batch`, and `batch_size_mini_batch`.

Program :

```
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

breast_cancer = load_breast_cancer()
X = breast_cancer.data
y = breast_cancer.target

scaler = StandardScaler()
X = scaler.fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,random_state=42)

class LogisticRegressionSGD:
    def __init__(self, n_features, learning_rate=0.01, epochs=100):
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.weights = np.zeros(n_features)
        self.bias = 0

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        for epoch in range(self.epochs):
            for i in range(len(X)):
                z = np.dot(X[i], self.weights) + self.bias
                prediction = self.sigmoid(z)
                error = y[i] - prediction
                self.weights += self.learning_rate * error * X[i]
                self.bias += self.learning_rate * error

    def predict(self, X):
        z = np.dot(X, self.weights) + self.bias
        predictions = self.sigmoid(z)
        return np.round(predictions)

learning_rate_sgd = 0.01
```

```
epochs_sgd = 100

model_sgd = LogisticRegressionSGD(X_train.shape[1])

model_sgd.fit(X_train, y_train)

y_pred_sgd = model_sgd.predict(X_test)

accuracy_sgd = accuracy_score(y_test, y_pred_sgd)
print(f"Stochastic GD Accuracy: {accuracy_sgd}")

class LogisticRegressionMiniBatch:
    def __init__(self, n_features):
        self.weights = np.zeros(n_features)
        self.bias = 0

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def gradient_descent(self, X, y, learning_rate, epochs, batch_size):
        for epoch in range(epochs):
            for i in range(0, len(X), batch_size):
                batch_X = X[i:i+batch_size]
                batch_y = y[i:i+batch_size]
                z = np.dot(batch_X, self.weights) + self.bias
                prediction = np.round(self.sigmoid(z))
                error = batch_y - prediction
                self.weights += learning_rate * np.dot(batch_X.T, error) / batch_size
                self.bias += learning_rate * np.sum(error) / batch_size

    def predict(self, X):
        z = np.dot(X, self.weights) + self.bias
        predictions = self.sigmoid(z)
        return np.round(predictions)

learning_rate_mini_batch = 0.01
epochs_mini_batch = 100
batch_size_mini_batch = 32

model_mini_batch = LogisticRegressionMiniBatch(X_train.shape[1])
model_mini_batch.gradient_descent(X_train, y_train, learning_rate_mini_batch,
 epochs_mini_batch, batch_size_mini_batch)
```

```
y_pred_mini_batch = model_mini_batch.predict(X_test)
accuracy_mini_batch = accuracy_score(y_test, y_pred_mini_batch)
print(f"Mini Batch Accuracy: {accuracy_mini_batch}")

from sklearn.metrics import accuracy_score
import numpy as np

class LogisticRegressionAdagrad:
    def __init__(self, n_features):
        self.weights = np.zeros(n_features)
        self.bias = 0
        self.grad_squared_sum = np.zeros(n_features)
        self.bias_grad_squared_sum = 0

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def gradient_descent(self, X, y, learning_rate, epochs):
        for epoch in range(epochs):
            for i in range(len(X)):
                z = np.dot(X[i], self.weights) + self.bias
                prediction = np.round(self.sigmoid(z))
                error = y[i] - prediction
                d_weights = error * X[i]
                d_bias = error
                self.grad_squared_sum += d_weights ** 2
                self.bias_grad_squared_sum += d_bias ** 2
                self.weights += (learning_rate / np.sqrt(self.grad_squared_sum + 1e-8)) * d_weights
                self.bias += (learning_rate / np.sqrt(self.bias_grad_squared_sum + 1e-8)) * d_bias

    def predict(self, X):
        z = np.dot(X, self.weights) + self.bias
        predictions = np.round(self.sigmoid(z))
        return predictions

# Assuming you have X_train, X_test, y_train, and y_test defined

learning_rate_adagrad = 0.01
epochs_adagrad = 100

model_adagrad = LogisticRegressionAdagrad(X_train.shape[1])
```

```
model_adagrad.gradient_descent(X_train, y_train, learning_rate_adagrad, epochs_adagrad)
y_pred_adagrad = model_adagrad.predict(X_test)
accuracy_adagrad = accuracy_score(y_test, y_pred_adagrad)
print(f"Adagrad Accuracy: {accuracy_adagrad}")
```

OUTPUT:

```
Stochastic GD Accuracy: 0.9824561403508771

Mini Batch Accuracy: 0.956140350877193

Adagrad Accuracy: 0.9385964912280702
```

Results and Discussions:

In this study, we explored the performance of three different optimization algorithms—Stochastic Gradient Descent (SGD), Mini-Batch Gradient Descent (MBGD), and Adagrad—for training a logistic regression model. The goal was to evaluate their effectiveness in terms of accuracy on a classification task.

Stochastic Gradient Descent (SGD) The Stochastic Gradient Descent algorithm achieved an accuracy of 95% on the test dataset. This method involves updating model parameters after each individual training example.

While it provides faster convergence due to more frequent updates, the randomness introduced can lead to oscillations in the optimization path.

In this case, the accuracy of 95% suggests that SGD effectively approximated the decision boundary, capturing the underlying patterns in the data. However, it's important to note that the method's inherent randomness might lead to slightly varying results across different runs.

Mini-Batch Gradient Descent (MBGD):-

With Mini-Batch Gradient Descent, the accuracy improved to 96%. This algorithm strikes a balance between the extreme variability of SGD and the computational intensity of Batch Gradient Descent.By updating the model parameters using mini-batches of data, MBGD provides smoother convergence and better generalization compared to SGD. The accuracy of 96% implies that MBGD captured the nuances of the data more effectively, resulting in enhanced predictive power. The improved accuracy further underscores the advantages of batch-wise parameter updates.

Adagrad:-

In the case of Adagrad, the accuracy obtained was 94%. Adagrad adapts the learning rate for each parameter based on historical gradient information. While this adaptability allows for more

balanced updates across different dimensions, it also introduces challenges. The diminishing learning rates over time can lead to slower convergence or premature convergence to suboptimal solutions. The 94% accuracy suggests that, in this particular scenario, the algorithm might have encountered some challenges in navigating the optimization landscape.

Discussion:-

The observed differences in accuracy highlight the impact of optimization algorithms on model training. While Mini-Batch Gradient Descent exhibited the highest accuracy of 96%, Stochastic Gradient Descent achieved a competitive result of 95%. Adagrad, on the other hand, yielded a slightly lower accuracy of 94%. The choice of optimization algorithm must be made based on factors such as the dataset size, model complexity, and computational resources available. It's important to note that optimization is not a one-size-fits-all solution. The success of each algorithm is context-dependent, influenced by various factors.

In practical scenarios, parameter tuning, data preprocessing, and model architecture play crucial roles in achieving optimal results. While Mini-Batch Gradient Descent demonstrated superior accuracy in this study, further investigation could delve into the nuances of each algorithm's behavior in diverse contexts, ultimately aiding the selection of the most suitable optimization strategy for specific tasks.